Game Tree Search and Game Theory

Shamelessly lifted from Andrew W. Moore

by David Cohn

cohn+@cs.cmu.edu

http://www.cs.cmu.edu/~cohn

See Russell and Norvig Chapter 5

Overview

- Definition of games and game terminology
- Game trees and game-theoretic values
- Computing game-theoretic values with recursive minimax.
- Other ways to compute game-theoretic value: Dynamic Programming copes with stalemates.
- Alpha-beta algorithm (good news.. it's not really as fiddly as it looks)
- Playing games in real-time
- Non-determinism
- Hidden Information
- Matrix Forms of Games
- Game Theory

2-player zero-sum discrete finite deterministic games of perfect information

What do these terms mean?

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses. (Doesn't mean fairness: "On average, two equal players will win or lose equal amounts" not necessary for zero-sum)
- Discrete: All game states and decisions are discrete values.
- Finite: There are only a finite number of states and decisions.
- Deterministic: No chance, (no dice rolls).
- Games: See next page
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).

Definition

A Two-player zero-sum discrete finite deterministic game of perfect information is a quintuplet: (S , I , Succs , T , V) where

- S = A finite set of states (note: state includes information sufficient to deduce who is due to move)
- I = The initial state
- Succs is a function which takes a state as input and returns a set of states: the legal positions after a move. Succs is non-empty if its argument is a non-terminal state
- T is a subset of S. It is the terminal states: the set of states at which the game ends, and payoff occurs.
- V is a mapping from terminal states to real numbers. It is the payoff to player A and the loss to player B.

Convention: assume Player A moves first.

For convenience: Assume turns alternate.

Nim: informal description

You begin with a number of piles of matches.

In your turn you may remove any number of matches from one pile.

The last person to remove a match loses.

In II-Nim, you begin with two matches, each with two piles.

States of II-Nim:

(ii,ii)-A , (i,ii)-A , (_,ii)-A , (ii,i)-A , (i,i)-A, (_,i)-A, (_,ii)-A, (_,i)-A, (_,_)-A

(ii,ii)-B , (i,ii)-B , (_,ii)-B , (ii,i)-B , (i,i)-B, (_,i)-B , (_,ii)-B , (_,i)-B , (_,_)-B

A common trick: By symmetry, some of the states are trivially equivalent (e.g. (_,ii)-A and (ii,_)-A). Make them one state by some canonical description (e.g. left pile never larger than right).

Nim

S = { (ii,ii)-a , (i,ii)-a , (_,ii)-a , (i,i)-a , (_,i)-a , (_,_)-a , (ii,ii)-b , (i,ii)-b , (_,ii)-b , (i,i)-b , (_,i)-b , (_,_)-b}

I = (ii,ii)

Succs(ii,ii)-A = { (i,ii)-B , (_,ii)-B }

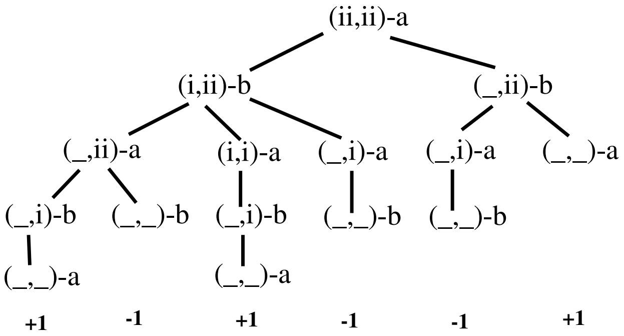
Succs(i,ii)-B = { (_,ii)-A , (i,i)-A , (_,i) }

etc ...

T = (_,_)-A , (_,_)-B

V(_,_)-A = +1 V(_,_)-B = -1

Game tree of II-Nim:

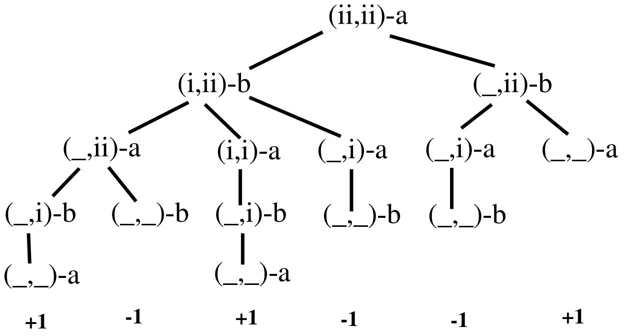


Important property of zero-sum games: all states can be given a value: not just the terminals. Uses the minimax definition..

Game theoretic value of Nim

Game theoretic value (also known as the minimax value) of a state is the value of a terminal that will be reached assuming both players use their optimal strategy.

Easy to fill in the II-nim tree bottom up to find minimax values of all states:



Are there other ways than "bottom-up-over-the-tree"? What are the computational issues like?

Minimax Algorithm

We've seen an algorithm: Generate the full game tree, and store it in memory. Run through all the terminal states, assigning values. Run through all predecessors assigning values. Etc.

Is it really necessary to explicitly store the whole tree in memory? Of course not. We can do the same trick that DFS uses.

```
Minimax-value(S) =
  If ( S is a terminal )
     return V(S)
  Else
     Let S1, S2, ... Sk = Succs(S)
     Let vi = Minimax-value(Si) foreach i
     If Player-to-move(State) = A
        return Max-over-i of vi
     else
        return Min-over-i of vi
```

Questions

- In the ordinary search problems, the algorithms were usually designed to return the path to the goal that they found. Our minimax algorithm doesn't bother. Why not? Does that matter?
- Recursive Minimax defined just now is a depth-first search algorithm. Would a breadth-first version be possible? How would it work?
- What if there are loops possible in the game?
  - Is our recursive-minimax guaranteed to succeed?
  - Is our recursive-minimax guaranteed to fail?
  - What problems do loops cause for our definition of minimax value (i.e. game-theoretic value)?
  - How could we fix our recursive minimax program?

Dynamic Programming

Say you have a game with N states. The length of the game is usually l moves. There are b successors of each state.

Minimax requires $O(b^l)$ states expanded.

This is best-case as well as worst-case (unlike DFS for simple search problems, which in best-case could be $O(l)$).

What if the number of states is smaller than b^l? e.g.. in chess, $b^l=10^{120}$, but N=a mere 10^{40}

Dynamic Programming is a better method in those cases, if you can afford the memory.

DP costs only $O(Nl)$.

DP for Chess Endgames

Suppose you only have, say, 4 pieces in total left on the board. With enough compute power you can compute, for all such positions, whether the position is a win for Black, White or a draw.

Assume N such positions.

1. With each state, associate an integer. A state code, so there's a 1-1 mapping between board positions and integers from 0...N-1.
2. Make a big array (2 bits per array entry) of size N. Each elemnent in the array may have one of three values:
   - ?: We don't know who wins from this state
   - W: We know white's won from here
   - B: We know black's won from here

DP Chess Endgames (ctd)

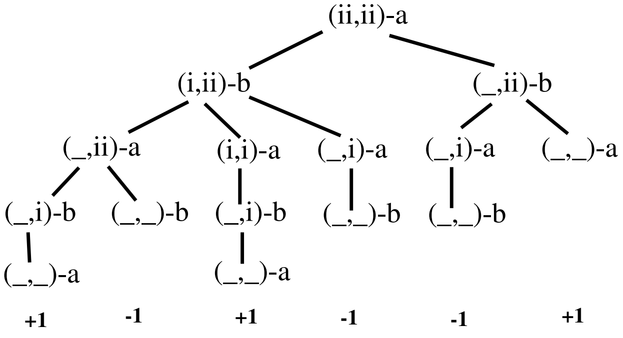
3. Mark all terminal states with their values (W or B).
4. Look through all states that remain marked with ?.

   For states in which W is about to move:
   - If all successor states are marked B, mark the current state as B.
   - If any successor state is marked W, mark the current state as W
   - Else leave current state unchanged.

   For states in which B is about to move:
   - If all successor states are marked W, mark the current state as W.
   - If any successor state is marked B, mark the current state as B
   - Else leave current state unchanged.
5. Goto 4, but stop when one whole iteration of 4 produces no changes.
6. Any state remaining at ? is a state from which no-one can force a win.

Cutting off un-needed parts of recursive Minimax

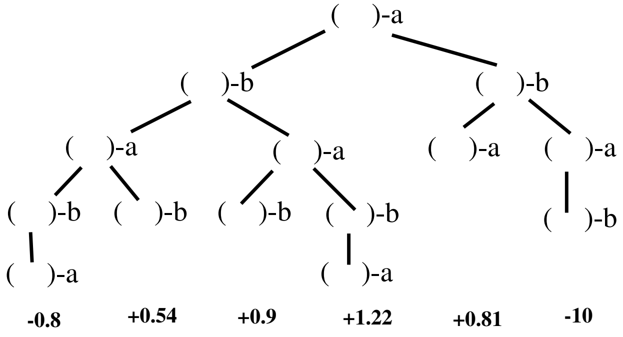
Suppose you knew that the only possible outcomes of the game were -1 and 1. What computation could be saved?



Answer: in general a lot (though not much here). If any successor is a forced win for the current player, don't bother with expanding further successors.

What if you didn't know the range of possible outcome values? We'll see that this is an important question.

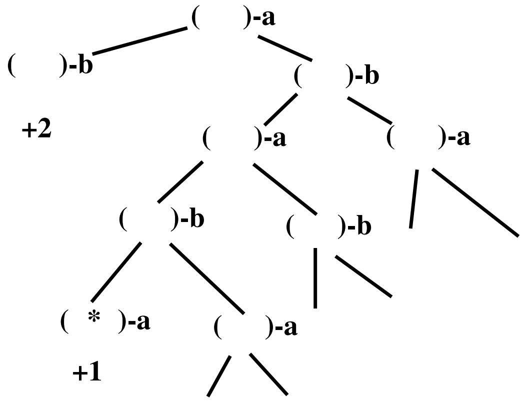
How can you cut-off with unknown terminal values?



Just do the depth first search as normal, but when you discover something that means your parent would definitely not choose you, don't bother with the rest of your successors.

In fact, it's not just your parent you should worry about, but any of your ancestors.

An ancestor causing cut-off

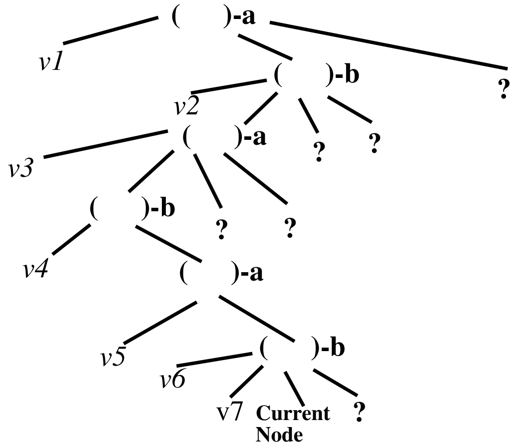


Suppose we've so far done a full depth first search, expanding left-most successors first, and have arrived at the node marked * (and discovered its value is +1).

What can we cut off in the rest of the search, and why?

General rule. We can be sure a node will not be visited if we're sure that either player has a better alternative at any ancestor of that node.

The general cutoff rule



In example: let $\alpha = \max(v1,v3,v5)$. If $\min(v6,v7) <= \alpha$, then we can be certain that it is worthless searching the tree from the current node or the sibling on its right.

In general: if at a B-move node, let α = max of all A's choices expanded on current path. let β = min of B's choices, including those at current node. Cutoff if $\beta <= \alpha$.

In general: Converse rule at an A-move node.

alpha-beta pruning* (from Russell)

```
function Max-Value (s,$\alpha$,$\beta$)
inputs:
   s: current state in game, A about to play
   $\alpha$: best score (highest) for A along path to s
   $\beta$: best score (lowest) for B along path to S
output: min( $\beta$ , best-score (for A) available from s)

   if ( s is a terminal state )
   then return ( terminal value of s )
   else for each s' in Succs(s)
            $\alpha$ := max( $\alpha$ , Min-value (s',$\alpha$,$\beta$) )
            if ( $\alpha$ >= $\beta$ ) then return $\beta$
      return $\alpha$

function Min-Value (s,$\alpha$,$\beta$)
output: max( $\alpha$ , best-score (for B) available from s)

   if ( s is a terminal state )
   then return(terminal value of S)
   else for each s' in Succs(s)
            $\beta$:= min( $\beta$, Max-value (s',$\alpha$,$\beta$) )
            if ( $\beta$ <= $\alpha$) then return $\alpha$
      return $\beta$
```

Assumes moves are alternate

How useful is alpha-beta?

What is the best possible case performance of alpha beta? Suppose that you were very lucky in the order in which you tried all the node successors. How much of the tree would you examine?

In the best case, the number of nodes you need to search in the tree is $O(b^{d/2})$... the square root of the recursive minimax cost.

Questions:

- How are alpha-beta and A* related, if at all? Could alpha-beta's trick help A* and could A*'s trick help alpha-beta?
- Does alpha-beta behave sensibly with loops?
- What can we do about large realsized games with huge numbers of states (e.g. chess)?

Game-Playing and Game-Solving

Two very different activities.

So far, we have been solely concerned with finding the true game-theoretic value of a state.

But what do real chess-playing programs do?

They have a couple of interesting features that the search and planning problems we've discussed to date on this course don't have:

- They cannot possibly find a guaranteed solution.
- They must make their decisions quickly, in real time.
- It is not possible to pre-compute a solution.

The overwhelmingly popular solution to these problems are the well-known *heuristic evaluation functions for games*.

Eval. functions in games

An evaluation function maps states to a number. The larger the number, the larger the true game-theoretic position is estimated to be.

- Search a tree as deeply as affordable.
- Leaves of the tree you search are not leaves of the game tree, but are instead intermediate nodes.
- The value assigned to the leaves are from the evaluation function.

Intuitions:

Visibility: the evaluation function will be more accurate nearer the end of the game, so worth using heuristic estimates from there.

Filtering: if we used the evaluation function without searching, we'd be using a handful of inaccurate estimates. By searching we are combining thousands of these estimates, hopefully eliminating noise.

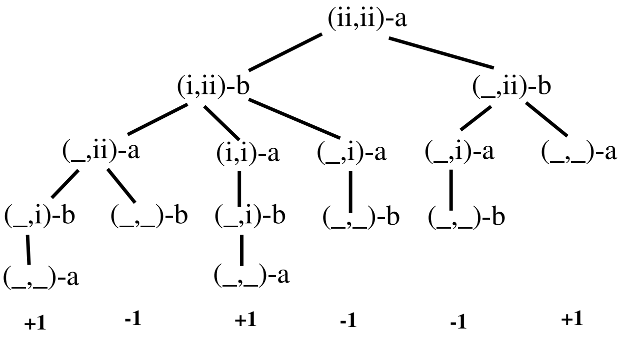
Dubious intuition. Counter-examples. But often works very very well in real games.

Other important issues for real game playing programs

- How to decide how far to search if you only have a fixed time to make a decision. What's the obvious sensible answer?
- Quiescence. What if you stop the search at a state where subsequent moves dramatically change the evaluation?

  The solution to the quiescence problem is a sensible technique called *quiescence search*.
- The horizon problem. What if *s* is a state which is clearly bad because your opponent will inevitably be able to do something bad to you? But you have some delaying tactics. The search algorithm won't recognize the state's badness if the number of delaying moves exceeds the search horizon.
- Endgames are easy to play well. How?
- Openings fairly easy to play well. How?

What if you think you're certainly going to lose?



What should A do in this situation?

What heuristics/assumptions could be used to cause A to make that decision? Two common methods.

Solving Games

Solving a game means proving the game-theoretic value of the start state.

Some games have been solved. Usually by brute force dynamic programming. (e.g. Four-in-a-row, many chess endgames)

Or brute force dynamic programming back from end of game, to create an end-game database, in combination with alpha-beta search from the start of the game. (Nine men's morris)

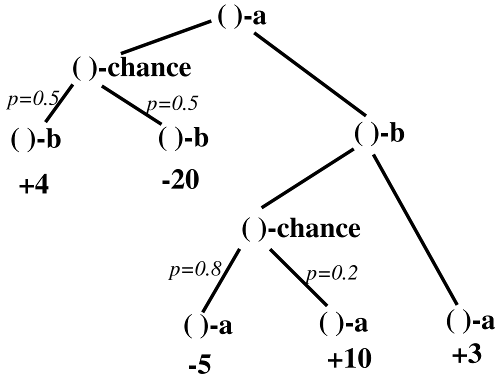
Or mostly brute force, with some game specific analysis (Connect-4).

Checkers may not be far off from being solved.

Solving a game is often very different from playing well at the game.

2 player zero-sum finite NONdeterministic games of perfect information

The search tree now includes states where neither player makes a choice, but instead a random decision is made according to a known set of outcome probabilities.



Game theory value of a state is the expected final value if both players are optimal.

If no loops, computing this is almost as easy as recursive minimax. Is there alpha-beta version?

Games with Hidden Information

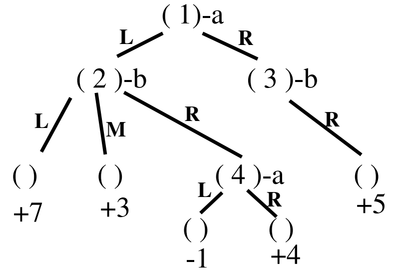
For extra reading, see an elementary Game Theory text such as the first chapter of the book "GAME THEORY: Mathematical Models of Conflict" by A J Jones, (Ellis Horwood, 1980)

Overview

- Games of Hidden Information: Pure and Mixed strategies and "bluffing".
- Is this game-playing stuff useful?
- Discussion

Game Theory

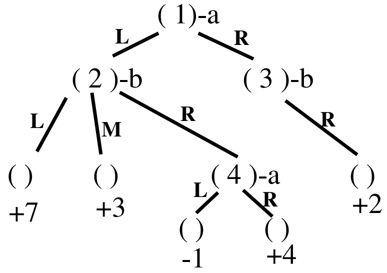
A **pure strategy** for a player is a mapping from all the possible states that player could see to the move the player would make.



Four pure strategies for A:
A's Strategy I: (1->L , 4->L)
A's Strategy II: (1->L , 4->R)
A's Strategy III: (1->R , 4->L)
A's Strategy IV : (1->R , 4->R)
Three pure strategies for B:
B's Strategy I: (2->L , 3->R)
B's Strategy II: (2->M , 3->R)
B's Strategy III: (2->R , 3->R)

In general, if each player can see N possible states, and there are b moves from each state, how many pure strategies?

Matrix forms of games



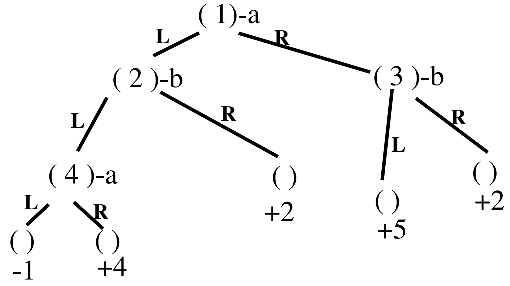
A's Strategy I: (1->L , 4->L)
A's Strategy II: (1->L , 4->R)
A's Strategy III: (1->R , 4->L)
A's Strategy IV : (1->R , 4->R)
B's Strategy I: (2->L , 3->R)
B's Strategy II: (2->M , 3->R)
B's Strategy III: (2->R , 3->R)

The matrix form shows the game value as a table indexed by A's and B's strategies.

	B-I	B-II	B-III
A-I	7	3	−1
A-II	7	3	4
A-III	2	2	2
A-IV	2	2	2

This matrix comprises a full description of the game! The rules etc are superfluous details!

Matrix Form Example



How many pure strategies does a have?

How many pure strategies does b have?

What is the shape of a Matrix form?

What is a matrix form of the above game?

Minimax with Matrix Forms

$$\begin{array}{c|ccc} & \text{B-I} & \text{B-II} & \text{B-III} \\ \hline \text{A-I} & 7 & 3 & -1 \\ \text{A-II} & 7 & 3 & 4 \\ \text{A-III} & 2 & 2 & 2 \\ \text{A-IV} & 2 & 2 & 2 \end{array}$$

A can decide from this matrix which strategy is best. For each strategy A considers the worst-case counter strategy by B. A chooses the row with the maximum minimum value. For A, the value of the game is this value.

In this example A chooses A-II, and says game has value 3.

When B decides which strategy is best, B searches for which column has the minimum maximum value.

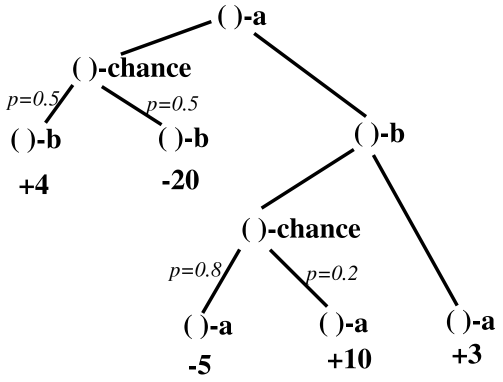
In this example, B chooses B-II, and says game has value 3.

Fundamental game theory result (proved by von Neuman):

> *In a 2-player, zero-sum game of perfect information, Minimax == Maximin. And there always exists an optimal pure strategy for each player.*

2 player zero-sum finite NONdeterministic games of perfect information

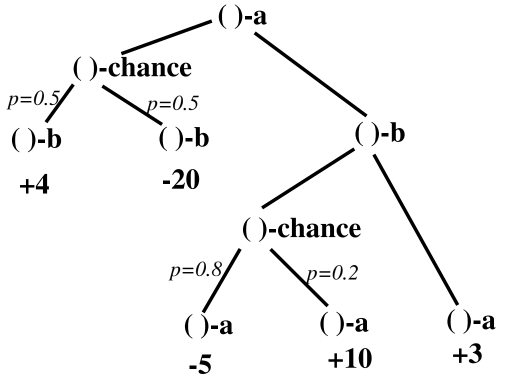
The search tree now includes states where neither player makes a choice, but instead a random decision is made according to a known set of outcome probabilities.



Game theory value of a state is the expected final value if both players are optimal.

Let's compute a matrix form of this!

NONdeterministic finite games: matrix forms



A's strategy I: (L), A's strategy II: (R)

B's strategy I: (L), B's strategy II: (R)

$$\begin{array}{c|cc} & \text{B-I} & \text{B-II} \\ \hline \text{A-I} & -8 & -8 \\ \text{A-II} & -2 & 3 \end{array}$$

The i,j'th entry holds the **expected** amount A will win if A follows its i'th strategy and B it's j'th. von Neuman's result still holds. Minimax = Maximin.

Two person zero-sum finite games, hidden information

Examples: Poker, 2-player bridge, Scrabble, Diplomacy. This gets very interesting.

Bottom line: Foundations of economic theory and "multiple agent" decision-making start here.

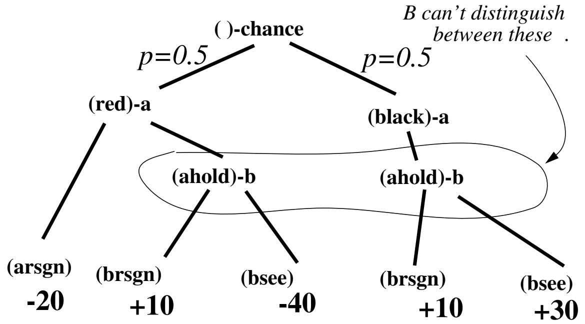
Bad news: the computational complexity of these foundations makes chess look like integer multiplication.

Imagine a version of mini-poker in which Red cards are bad for A and Black cards are good.

Player A is dealt a card. It is red or black with 50% probability.

A may resign if the card is red: *A loses 20c*
Else A "holds".
  B may then resign: *A wins 10c*
  B may "see":
    If card is red: *A loses 40c*
    If card is black: *A wins 30c*

Mini-Poker Pure Strategies



In games of hidden information, pure strategies are mappings from *all possible states that the player can detect* to moves.

For Player A there are two pure strategies:

- **Strategy RESIGNER:** *Resign if card = Red*
- **Strategy HOLDER:** *Hold if card = Red*

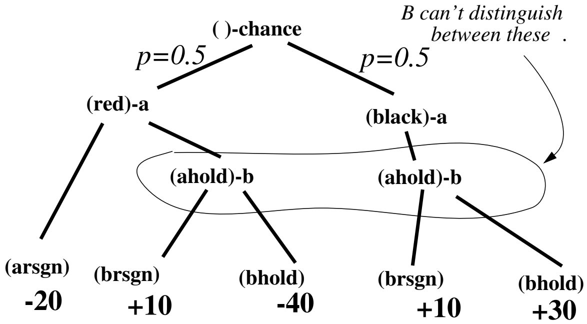
For Player B there are two pure strategies:

- **Strategy RESIGNER:** *Resign if A holds*
- **Strategy SEER:** *If A holds then see*

If A is a resigner, how much will A win on average? Depends on B's strategy!

If B is a resigner, how much will A win on average? Depends on A's strategy.

Mini-Poker in Matrix Form



The matrix form of a game shows *expected pay-offs to A* as it depends on A's and B's strategies:

	B-resigner	B-seer
A-resigner	–5	+5
A-holder	+10	-5

With hidden information, Minimax no longer necessarily equals Maximin.

A's optimal strategy depends on B's and vice versa. What can we do? What should a computer program for playing A do?

Von Neuman's game theoretic values for games of hidden information

It turns out for the game of mini-poker, the game theoretic value for A is 1c. A can expect to win 1 cent per game on average if A does the right thing.

Furthermore, A can even tell B in advance what A's strategy is. That information will not help B!

Mixed Strategies

The trick is that A must not use a pure strategy, but a **mixed strategy**, in which at the start of the game (before play) A selects which pure strategy it will use for that game at random. And there will be an optimal probability distribution for it.

For mini-poker, A must decide to be a holder with probability *p* and a resigner with probability *1-p*. How do we compute the optimal *p*?

Computing a mixed strategy for A: Guess number one: p=2/3

	B-resigner	B-seer
A-resigner	–5	+5
A-holder	+10	-5

Suppose A chooses to be a holder with probability p = 2/3. Suppose B knows this.

Which strategy will B pick?

How much will A win on average?

Computing a mixed strategy for A: Guess number two: p=1/3

	B-resigner	B-seer
A-resigner	–5	+5
A-holder	+10	-5

Suppose A chooses to be a holder with probability p = 1/3. Suppose B knows this.

Which strategy will B pick?

How much will A win on average?

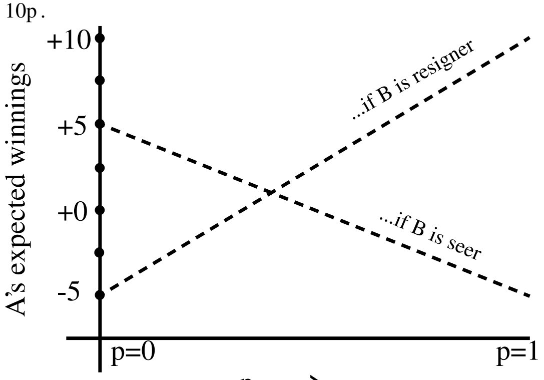
Computing a mixed strategy for A: All Guesses

	B-resigner	B-seer
A-resigner	–5	+5
A-holder	+10	-5

Suppose A chooses to be a holder with probability p.

If B uses the pure strategy of resigner, A's expected profit is 15p - 5.

If B uses the pure strategy of seer, A's expected profit is 5 - 10p.



Suppose A knows B will always use the most annoying pure strategy. What p should A use?

A's minimax optimal mixed strategy

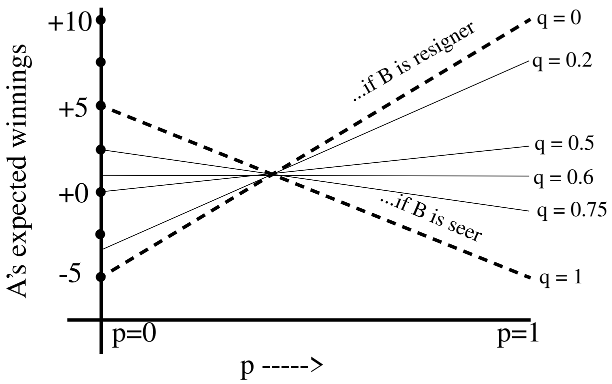
	B-resigner	B-seer
A-resigner	–5	+5
A-holder	+10	-5

The point where the two lines (15p - 5 and 5 - 10p) meet is where p = 0.4.

And then the expected profit is 1 cent if B is a seer.

And the expected profit is 1 cent if B is a resigner.

What if B uses a mixed strategy (choose seer with prob q and resigner with prob 1-q)?



Whatever B's mixed strategy, A is still guaranteed 1 cent.

Computing A's optimal mixed strategy for a 2x2 game

(2x2 game = game with a matrix form in which A and B each have two strategies).

$$\begin{array}{ccc} & \text{B-I} & \text{B-II} \\ \text{A-I} & m_{11} & m_{12} \\ \text{A-II} & m_{21} & m_{22} \end{array}$$

Recipe for computing A's optimal mixed strategy for a nXm game

(nXm game = game with a matrix form in which A has n pure strategies and B has m.)

Say Player A will use strategy *1* with prob p_1.
Say Player A will use strategy *2* with prob p_2.